A Component Data-Focused Method to Build the Executable Model for a DoDAF Compliant Architecture

Li Huang, Guangqi Huang, Yaohong Zhang, Weizi Li, Xueshan Luo

A Component Data-Focused Method to Build the Executable Model for a DoDAF Compliant Architecture

Li Huang[1, 2], Guang-qi Huang[1, 2], Yao-hong Zhang[1], Wei-zi Li[2], Xue-shan Luo[1]

[1] Science and Technology on Information Systems Engineering Laboratory, National University of Defense Technology,
410073 Changsha, China

[2] Informatics Research Centre, University of Reading, Whiteknights,
RG6 6UD Reading, UK
nudthli@gmail.com, cscsyz@163.com, zhang_yaohong@263.net, weizi.li@henley.ac.uk, luoxueshan@gmail.com

Abstract. An executable model plays an important role on verifying the behavior and performance of an architecture. This paper summarizes the state of the art on the synthesis methods of the executable model for an architecture that is compliant with the Department of Defense Architecture Framework (DoDAF). To overcome the deficiencies of current executable modeling studies, a component data-focused method is proposed. Following the introduction of an executable modeling language named OPDL, the data elements of DoDAF Meta-model (DM2) required for building executable model is analyzed. The mapping relations between partial DM2 and OPDL elements are built. Finally, a process to create executable model is explained in detail.

Keywords: Architecture, Architecture Verification, Executable Model, DoDAF, DM2

1 Introduction

The Department of Defense (DoD) Architecture Framework (DoDAF) [1] is the overarching, comprehensive framework and conceptual model enabling the development of architectures in the DoD. A DoDAF compliant architecture is presented by dividing all kinds of architecture descriptive data into manageable pieces, according to the stakeholder's viewpoint. Furthermore, data pertained to each viewpoint are described by a few predefined models respectively.

In order to verify the behavior and performance of the architecture and address the concerns of the customer, Levis and Wagenhals [2] proposed that an executable model which is synthesized from architecture models is needed. An executable model of architecture enables the architect to analyze the dynamic behaviors of the architecture, identify logical and behavioral errors that are not easily seen in the static descriptions of its models, and demonstrate the capabilities of the architecture to the users.

Wagenhals et al. [3-5] use Colored Petri Nets (CPN) language for creating the executable model of the architecture. The approaches have also been developed that allow the derivation of CPN model of an architecture designed by either structured analysis or object-oriented methodology. Wang et al. [6] proposes the method of transforming System Modeling Language (SysML) based architecture models to CPN. Baumgarten and Silverman [7] converted several architecture models to ExtendSim model which enables workload, timing, and process analysis that can help identify gaps, bottlenecks and overloads to queues. Mittal, Zeigler, et al. [8-9] presented extensions to the DoDAF to support specification of DoDAF architectures within a development environment based on Discrete Event System Specification (DEVS) model, and demonstrated how DoDAF-DEVS model mapping can actually take place from the existing DoDAF UML specifications. As part of the Architecture-based Technology Evaluation and Capability Tradeoff (ARCHITECT) methodology developed by Griendling and Mavris [10], an approach adopting a standard set of DoDAF models and the associated data to create four types of executables models is detailed.

Overall, the aforementioned executable modeling studies have the following characteristics: 1) every method has its specific executable modeling languages, most of which are some extensions to original Petri net. 2) every method contains algorithms to transform static architectural models described by different methodologies into executable models. These algorithms are generally different in various architecture methodologies due to the emphasis on architectural model instead of the underlying architecture data, which is one of the deficiencies of previous version of DoDAF. 3) most methods focus on validating the correctness of the logic and behavior of architecture, hence they usually extract much information from activity or state related models in the architecture to build executable models, but little information from component related models.

Methodology-specific executable model conversion algorithms need architect to be familiar with both original architectural modeling languages and target executable formalisms, and it is also difficult to be commonly understood and compared across multiple instances [11]. Hence, it is almost impossible to popularize these methods widely. As the latest DoDAF version 2.0 [12] has shifted to a “data-centric” approach by building the DoDAF Meta-model (DM2), it places greater emphasis on architecture data as the necessary ingredient for architecture development. DM2 defines architectural data elements, their associations and attributes, thus providing a high-level view of the data normally collected, organized, and maintained in an architectural description effort. It is feasible to make research on new executable model synthetic methods which are architectural modeling methodology-independent, such as those done by [11] and [13].

But the executable models built by these new studies still lack system component information. System component information plays an important role on the verification of architectural behavior and performance, which are affected by architectural components and structure. The lack of system component information in the generated executable models will constrain their ability in evaluating more user concerns such as system structural fitness, measures of performance and measures of effectiveness of system, and etc.

In order to solve the issues mentioned above, we propose a component data-focused method to build the executable model for a DoDAF based architecture. It will build the executable model upon DM2, which is architectural modeling methodology-independent. At the same time, component information will be extracted from the underlying data of architecture models and embodied in the generated executable model as a main part. As to the selection of executable modeling language, we select the Object Petri Nets based Description Language (OPDL) [14], which was developed by our laboratory. OPDL is an extension of original Petri net. It offers an advantage of combining a well-defined mathematical foundation, the graphical representation and interactive simulation capabilities to check both the logical and functional correctness of a system and to make performance analysis. Many architectural elements, e.g. information, materiel, and data can be defined by different types of tokens in OPDL. These features make OPDL flexible and capable of modeling complex systems. Furthermore, OPDL is simple to use, whereas CPN is relatively more complicated.

The remainder of this paper is organized as follows. Section 2 gives a brief introduction of OPDL. Section 3 examines DM2 data elements which are related to component information for building executable model. In the meantime the mapping relations between DM2 and OPDL elements are established. Section 4 presents the steps for building OPDL based executable model.

2 OPDL

To implement object model in the original Petri net, an Object Petri Nets based Description Language (OPDL) is designed [14]. OPDL extents Petri nets on two aspects. First, it introduces object-oriented methodology into Petri nets to form Object Petri Nets (OPN), where the object is the basic modeling unit and reusable module. Second, OPDL adds new model elements to original Petri net, such as switch, and attributions to all of the model elements.

Below is a simple explanation for the basic model elements of OPDL.

Class. An OPN class includes four parts, which are property table, OPN description, initialization function and post-instancing function. 1) Property table is a data space used by the object. Each item in the table is composed of a property name and a property value. A property value can be accessed via its name. 2) OPN description is an OPN graph. Different graphs connect together through their input ports and output ports. 3) Initialization function is used to initialize the object instance when it is created, such as setting the initial value of a property. 4) Post-instancing function will be executed after the object instance has been created. All the classes are stored in a class library. Modeler builds and organizes model based on the class library. Class quotation mechanism supports the reuse of model. Hence, modeling is the process of designing and using classes.

Object. An OPN object is an instance of some class. The attributions of an object can be modified by its instance function. There are two sorts of relations between two different objects, i.e. interactive relation and nesting relation. By interactive relation, two objects connect together via their input ports and output ports. By nesting relation, two objects form hierarchy model.

Place. A place is a kind of data structure, which has a queue to buffer tokens. Each place corresponds to a color. Only those tokens which have the same color can enter into the place. The attributions of a place include rule of queue, token capacity and the event processing function. When a token enters or gets out the place, its event processing function will be invoked. OPDL also defines a special kind of place named port. A port has the same attributions as a place. Ports are divided into input ports and output ports. The successor of an output port should be an input port, and the predecessor of an input port should be an output port.

Transition. The attributions of a transition include priority, delay function, predicate function, action function, and event processing function. Priority is used to handle conflicts. Delay function is used to determine the process time of a transition. Predicate function is a condition which needs to be satisfied for a transition to be enabled. Action function is a sequence of operations that a transition will carry out after it fires. When a transition begins to fire, the event processing function will be executed.

Switch. A switch can be regarded as a special kind of transition. Unlike a common transition, it does not distribute any tokens automatically at the end of its firing, and it is up to the modeler to determine how to distribute the tokens to its successive places.

Arc. An Arc is used to connect the place and the transition just as the original Petri nets. Moreover, it is used to connect an input port and an output port.

Token. A token is defined as a structural data and corresponds to a color. A token has a property table. In the functions of a transition, the items of property table can be inserted and accessed.

To implement OPDL, we developed a software tool named OPMSE [14], which is a kind of integrated modeling and simulation environment.

3 DM2 Data Elements Required for Building Executable Model

As mentioned above, the aim of our method is to create an executable model that can be used to verify system structural fitness, measures of performance and measures of effectiveness of system, and so on. Thereby, component information of the whole architecture which describes a system is required for building the executable model. The data elements in DM2 related to component information need to be examined. Then, the mapping relations between DM2 and OPDL elements need to be established to create OPDL based executable model.

3.1 DM2 Data Elements Related to System Component

The essence of DM2 is to answer the set of standard interrogatives, which are the set of questions, Who, What, When, Where, Why, and How [1]. Accordingly, data elements in DM2 is divided into 12 categories of data groups, which are Performers, Resource Flows, Information and Data, Activities, Training/Skill/Education, Capability, Services, Projects, Goals, Rules, Measures, and Locations.

We begin by finding the data element that is used to represent system component information. It is easy to determine that the Performers data group answers the Who

question, and its *System* data element represents system component information. Then, the data element that represents system function information is *Activity*, which is in the Activities data group. A *System* may have several *Activities*. And also, a system function transforms input information into output information, which is represented by *Information* data element in the Information and Data group. A system may have several performance parameters, which are represented by *Measure* data element in the Measures data group. Thus, *System*, *Activity*, *Information* and *Measure* form the basic DM2 data elements related to system component, as shown in Fig. 1.

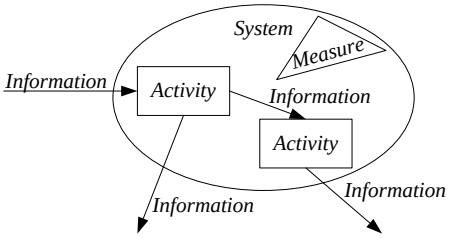

Fig. 1. DM2 data elements related to system component

Besides data elements, the associations between them need to be considered. We can find these associations which are *activityPerformedByPerformer*, *activityResourceOverlap*, *wholePartType*, and *measureOfTypeResource*. The *activityPerformedByPerformer* association represents that an *Activity* is performed by a *System*. The *activityResourceOverlap* association represents that a piece of *Information* is produced by an *Activity* and consumed by another *Activity*. The *wholePartType* association represents that a *System* is a part of another *System*. The *measureOfTypeResource* association represents that a *Measure* belongs to a *System*.

According to DM2, architecture data can be saved in XML format. The figures from Fig. 2 to Fig. 6 illustrate some architecture data segments related to system component information of a notional system.

```
- <System ideas:FoundationCategory="IndividualType" id="s2">
  <ideas:Name exemplarText="System A" namingScheme="ns1" id="n89" />
  </System>
- <System ideas:FoundationCategory="IndividualType" id="s3">
  <ideas:Name exemplarText="System B" namingScheme="ns1" id="n90" />
  </System>
- <System ideas:FoundationCategory="IndividualType" id="s4">
  <ideas:Name exemplarText="System C" namingScheme="ns1" id="n91" />
  </System>
- <System ideas:FoundationCategory="IndividualType" id="s5">
  <ideas:Name exemplarText="System D" namingScheme="ns1" id="n92" />
  </System>
```

Fig. 2. *System* segment of architecture data

```
- <Activity ideas:FoundationCategory="IndividualType" id="a18">
  <ideas:Name exemplarText="Act A" namingScheme="ns1" id="n74" />
  </Activity>
- <Activity ideas:FoundationCategory="IndividualType" id="a19">
  <ideas:Name exemplarText="Act B" namingScheme="ns1" id="n75" />
  </Activity>
- <Activity ideas:FoundationCategory="IndividualType" id="a20">
  <ideas:Name exemplarText="Act C" namingScheme="ns1" id="n76" />
  </Activity>
- <Activity ideas:FoundationCategory="IndividualType" id="a21">
  <ideas:Name exemplarText="Act D" namingScheme="ns1" id="n77" />
  </Activity>
```

Fig. 3. *Activity* segment of architecture data

```
- <Information ideas:FoundationCategory="IndividualType" id="di14">
  <ideas:Name exemplarText="Info A" namingScheme="ns1" id="n84" />
  </Information>
- <Information ideas:FoundationCategory="IndividualType" id="di15">
  <ideas:Name exemplarText="Info B" namingScheme="ns1" id="n85" />
  </Information>
- <Information ideas:FoundationCategory="IndividualType" id="di16">
  <ideas:Name exemplarText="Info C" namingScheme="ns1" id="n86" />
  </Information>
- <Information ideas:FoundationCategory="IndividualType" id="di17">
  <ideas:Name exemplarText="Info D" namingScheme="ns1" id="n87" />
  </Information>
```

Fig. 4. Information segment of architecture data

```
<activityPerformedByPerformer ideas:FoundationCategory="TripleType"
id="app19" place1Type="s2" place2Type="a18" place3Type="apuc17" />
<activityPerformedByPerformer ideas:FoundationCategory="TripleType"
id="app20" place1Type="s2" place2Type="a19" place3Type="apuc18" />
<activityPerformedByPerformer ideas:FoundationCategory="TripleType"
id="app21" place1Type="s3" place2Type="a20" place3Type="apuc19" />
<activityPerformedByPerformer ideas:FoundationCategory="TripleType"
id="app22" place1Type="s3" place2Type="a21" place3Type="apuc20" />
```

Fig. 5. *activityPerformedByPerformer* segment of architecture data

```
<activityResourceOverlap ideas:FoundationCategory="TripleType"
id="aro14" place1Type="a18" place3Type="a20" place2Type="di14" />
<activityResourceOverlap ideas:FoundationCategory="TripleType"
id="aro15" place1Type="a18" place3Type="a21" place2Type="di15" />
<activityResourceOverlap ideas:FoundationCategory="TripleType"
id="aro16" place1Type="a19" place3Type="a22" place2Type="di16" />
<activityResourceOverlap ideas:FoundationCategory="TripleType"
id="aro17" place1Type="a18" place3Type="a19" place2Type="di17" />
```

Fig. 6. *activityResourceOverlap* segment of architecture data

3.2 Mapping relations between DM2 and OPDL elements

After the analysis of DM2 data elements related to system component information, the mapping relations between DM2 and OPDL elements is established according to their features, which is shown in Table 1.

Table 1. Elements mapping between partial DM2 and OPDL

DM2 data elements		OPDL model elements
data elements	*System*	class, object
	Activity	transition
	Information	place (input port, output port) , property of class
	Measure	property of class
associations between data elements	*activityPerformedByPerformer*	transition of class's OPN graph
	activityResourceOverlap	arcs
	wholePartType	constituent relation between two objects
	measureOfTypeResource	property of certain class

Each *System* can be mapped to an object in OPDL. For those objects have the same properties and OPN graphs, a class can be defined. If a *wholePartType* association exists between two *Systems*, then a parent-child relation exists between their corresponding objects.

Each *Activity* can be mapped to a transition in OPDL. If an *activityPerformedByPerformer* association exists between an *Activity* and a *System*, it means that the corresponding transition is part of the corresponding class's OPN graph.

Each piece of *Information* can be mapped to a place, an input port, an output port, or a property of class in OPDL. If an *activityResourceOverlap* association exists between two *Activities* and a piece of *Information*, then two arcs are needed to connect between each corresponding transition and the corresponding place or port.

Each *Measure* can be mapped to a property of class. If a *measureOfTypeResource* association exists between a *Measure* and a *System*, it means that the corresponding property belongs to the corresponding class.

4 Process of Building Executable Model

A process to build executable model is shown in Fig. 7. Firstly, we extract part information from architecture data that is saved following DM2, including *System*, *Activity*, *Information*, *Measure* and their associations. Then convert into their OPDL counterparts according to above established mapping relations. Some OPN classes will be created at this step, which we call initial OPN classes. Secondly, depending on the simulation purposes of executable model, some edits on these initial OPN classes need to be done, such as adding new places and transitions in their OPN graphs, initializing properties of classes in their initialization functions and post-instancing

functions, and defining the process time and operations of transitions in their delay functions and action functions respectively. All the OPN classes will be ready and organized in an OPN class library at this step. Lastly, we instantiate the top level OPN class in the library, which will also instantiate its descendent objects, and an executable model will be produced.

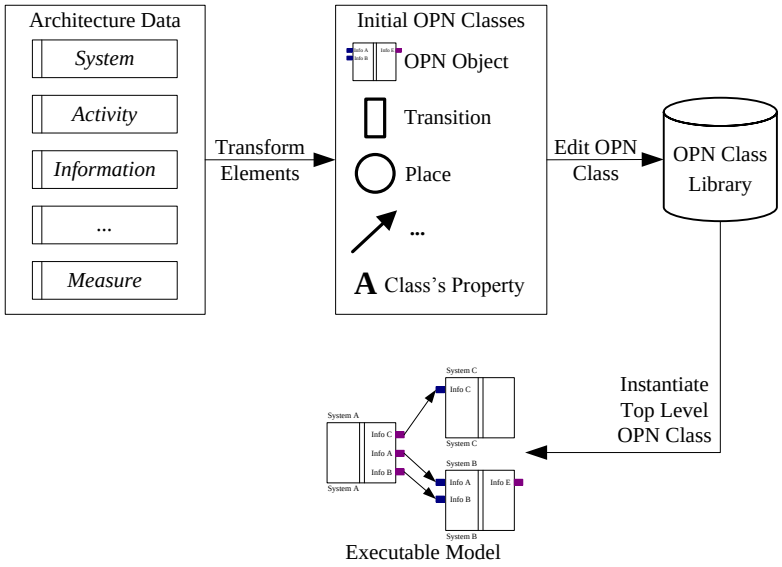

Fig. 7. A process to build executable model

The first step in above process can be subdivided into several sub steps as follows.

(1) Determination of OPN objects. For each *System* in architecture data, we define an object with the same name.

(2) Creation of OPN classes. For each object with particular features, a class is created. For those objects have similar features, we only create one class for them. Additionally, a top level class is always created in view of the modeling mechanism of OPDL. For composite classes determined by *wholePartType* association, their child objects are built into their OPN graphs.

(3) Creation of graph elements for OPN classes. Each *activityResourceOverlap* association involves three elements, i.e. an *Activity* which produces information, a piece of *Information*, and an *Activity* which consumes information. The *System* which performs activity to produce or consume information can be determined by *activityPerformedByPerformer* association. For each *Activity* that produces or consumes information, we create a transition in the class which is the counterpart of *System* that produce or consume information. For an *activityResourceOverlap* association, if it is the same *System* that produce and consume information, we create one place in the class; otherwise, we create an output port in one class and an input port in another class. Two arcs are built to connect two pairs of transitions and places

respectively. Arcs connecting the ports of child objects of a composite class need to be created based on *activityResourceOverlap* association too.

(4) Creation of properties for OPN classes. *Measures* in *measureOfTypeResource* association and *Information* in *activityResourceOverlap* association can be used to define candidate properties for class.

According to above process, we can build an OPN class library based on the part architecture data shown in Fig. 2 to Fig. 6. The OPN graph of top level class in the library is shown in Fig. 8. Fig. 9 illustrates the OPN graph of class "System A". We can get an executable model by instantiating the top level OPN class in OPMSE.

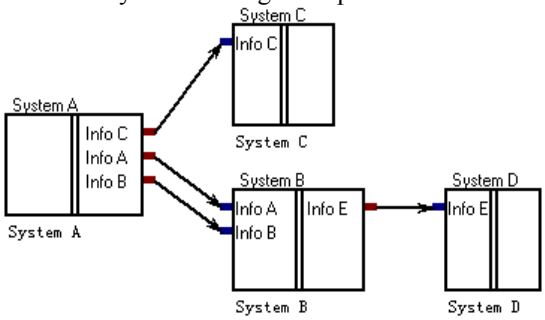

Fig. 8. The OPN graph of top level class

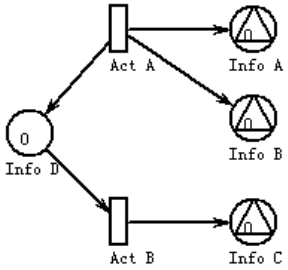

Fig. 9. The OPN graph of class "System A"

5 Conclusion

This paper presents a component data-focused method to synthesize the executable model for an architecture that is described compliant with the DoDAF. It is architectural modeling methodology-independent and is fit for the verification of architectural behavior and performance. Future research will involve adding additional data elements which are related to service, rule, and communication. This helps the transformation from architecture data into the executable model to enable more widely evaluation on user concerns. At the same time, a method to enable automatic executable model generation will also be studied.

Acknowledgments. This research was supported in part by the State Scholarship Fund (No. 201206115006) from China Scholarship Council.

References

1. DoD Architecture Framework Working Group.: DoD Architecture Framework version 2.0 Volume 1: Introduction, Overview, and Concepts. Department of Defense, Washington (2009)
2. Levis, A.H. and Wagenhals, L.W.: C4ISR Architectures I : Developing a Process for C4ISR Architecture Design. Systems Engineering. 3, 225-247 (2000)
3. Wagenhals, L.W., Shin, I., Kim, D. and Levis, A.H.: C4ISR Architectures II : Structured Analysis Approach for Architecture Design. Systems Engineering. 3, 248-287 (2000)
4. Wagenhals, L.W., Haider, S., Levis, A.H.: Synthesizing Executable Models of Object Oriented Architectures. Systems Engineering. 6, 266-300 (2003)
5. Wagenhals, L.W., Liles, S.W., Levis, A.H.: Toward Executable Architectures to Support Evaluation. In: The 2009 International Symposium on Collaborative Technologies and Systems, pp. 502-511. IEEE Press, New York (2009)
6. Wang, R.Z., Dagli, C.H.: An Executable System Architecture Approach to Discrete Events System Modeling Using SysML in Conjunction with Colored Petri Net. In: 2nd Annual IEEE Systems Conference, pp. 1-8. IEEE Press, New York (2008)
7. Baumgarten, E., Silverman S.J.: Dynamic DoDAF and Executable Architectures. In: 2007 Military Communications Conference, pp. 1-5. IEEE Press, New York (2007)
8. Mittal, S.: Extending DoDAF to Allow Integrated DEVS-based Modeling and Simulation. The Journal of Defense Modeling and Simulation. 3, 95–123 (2006)
9. Zeigler, B.P., Mittal, S.: Enhancing DoDAF with a DEVS-based System Lifecycle Development Process. In: 2005 IEEE International Conference on Systems, Man and Cybernetics, pp. 3244-3251. IEEE Press, New York (2005)
10. Griendling, K. and Mavris, D.N.: Development of a DoDAF-Based Executable Architecting Approach to Analyze System-of-Systems Alternatives. In: 2011 IEEE Aerospace Conference, pp. 1-15. IEEE Press, New York (2011)
11. Ge, B.F., Hipel K.W., Yang, K.W., Chen, Y.W.: A Data-Centric Capability-Focused Approach for System-of-Systems Architecture Modeling and Analysis. Systems Engineering. 16, 363-377 (2013)
12. DoD Architecture Framework Working Group.: DoD Architecture Framework version 2.0 Volume 2: Architectural Data and Models. Department of Defense, Washington (2009)
13. Zhang, X.X., Luo, X.S., Luo, A.M.: Method of Architecture Executable Evaluation Based on DM2. In: 3rd International Conference on System Science, Engineering Design and Manufacturing Informatization, pp. 213-217. IEEE Press, New York (2012)
14. Luo, X.S., Qiu, D.S., Rao, X.H., Bao, W.D.: OPMSE: An Object Petri Nets based Modeling and Simulation Environment. In: Mortensen, K.H. (eds.): Tool Demonstrations of 21st International Conference on Application and Theory of Petri Nets. pp. 65-69. Denmark (2000)